Texture Synthesis from Photographs

Christian Eisenacher, Sylvain Lefebvre, Marc Stamminger

EUROGRAPHICS 2008 / G. Drettakis and R. Scopigno
(Guest Editors)

Volume 27 (2008), Number 2

Texture Synthesis From Photographs

C. Eisenacher[1,2] and S. Lefebvre [2] and M. Stamminger [1]

[1]FAU Erlangen, Germany
[2]REVES / INRIA Sophia-Antipolis, France

Abstract

The goal of texture synthesis is to generate an arbitrarily large high-quality texture from a small input sample. Generally, it is assumed that the input image is given as a flat, square piece of texture, thus it has to be carefully prepared from a picture taken under ideal conditions. Instead we would like to extract the input texture from any surface from within an arbitrary photograph. This introduces several challenges: Only parts of the photograph are covered with the texture of interest, perspective and scene geometry introduce distortions, and the texture is non-uniformly sampled during the capture process. This breaks many of the assumptions used for synthesis.
In this paper we combine a simple novel user interface with a generic per-pixel synthesis algorithm to achieve high-quality synthesis from a photograph. Our interface lets the user locally describe the geometry supporting the textures by combining rational Bézier patches. These are particularly well suited to describe curved surfaces under projection. Further, we extend per-pixel synthesis to account for arbitrary texture sparsity and distortion, both in the input image and in the synthesis output. Applications range from synthesizing textures directly from photographs to high-quality texture completion.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation – Digitizing and scanning;

1. Introduction

Texture synthesis from example has become an increasingly interesting tool. While computer graphics applications now require massive amounts of high-quality high-resolution textures, texture synthesis algorithms are able to quickly generate these textures from small example images. However, synthesis algorithms expect a square image representing a flat piece of texture as input. This requirement is sometimes hard to enforce: We may only have access to a single photograph showing the surface under an arbitrary viewpoint and with occlusions or holes. For some objects, obtaining a flat sample is impossible: An apple skin or a piece of tree bark cannot be flattened without introducing new distortions. We are thus interested in designing a method to synthesize a texture using any arbitrary photograph as input. Ideally, the user would simply indicate from which surface to synthesize, and the algorithm would be able to synthesize more of the same texture. This process is depicted Figure 1.

The main challenges are:

- The texture in the input image is distorted by the perspective view and the underlying geometry.
- The example surface may have an arbitrary outline as occluded areas need to be ignored. In other words, the set of pixels on which we have useful information is sparse.
- The texture likely appears with varying amount of detail in different parts of the image. A good example of this is a wall photographed at an angle: More details are visible in the foreground.
- We have no description or prior knowledge of the surface geometry. Nevertheless, we need to define pixel neighborhoods in order to perform efficient texture synthesis.

It is worth noting that some existing algorithms perform synthesis *into* a distorted space [YHBZ01, LH06]. However, to the best of our knowledge, no algorithm synthesizes *from* a distorted input – which implies different challenges.

Our problem setting shares similarities with the work on image completion [BVSO03, DCOY03, PSK06]. Image

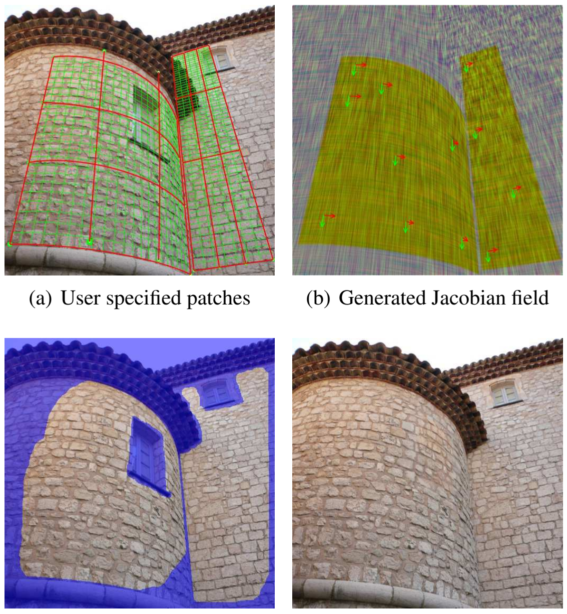

(a) User specified patches (b) Generated Jacobian field (c) Region of interest (ROI) (d) Synthesis result

Figure 1: *A user specifies patches to model texture distortions (a) as a Jacobian field (b). Analyzing texture from a user defined region of interest (c) we can directly synthesize from distorted space into distorted space (d).*

completion describes the filling of holes in images – for instance to remove an object from a photograph. Often, missing surfaces are seen at an angle and must be completed under distortion, from distorted content. Note however that we do *not* propose an image completion technique. Our goal is to design a texture synthesis algorithm that can use a texture under distortion as input and synthesize more of this texture under any new distortion, may it be a flat piece of toroidal texture, another photograph, or a texture atlas.

The first problem that needs to be tackled is to obtain some knowledge about the surfaces present in the image. We propose a simple user-guided approach based on the manual specification of a few curved patches covering the surface of interest. This is detailed in Section 3.1. Note that our texture synthesis scheme is not specialized to our user interface: The required information about the surfaces could possibly be obtained from a reconstruction technique.

The second major challenge is to adapt texture synthesis to exploit this information. We build our work on the fast per-pixel synthesis algorithm of Lefebvre and Hoppe [LH06]. Our contributions are to adapt the analysis step of the algorithm to take into account distortion, non-uniform sampling and sparsity of the input texture data. We also improve synthesis quality on anisotropic textures such as brick walls or tree barks. These contributions are detailed Section 3.2 and Section 3.3.

2. Related work

Traditional example based texture synthesis algorithms synthesize a large flat piece of texture from a small input example. Two main approaches exist: Per-pixel algorithms generate the texture pixel by pixel [EL99, WL00], while patch-based algorithms cut and paste together large patches to form the new texture [EF01, KSE*03]. Both have many extensions, in particular to perform synthesis over a surface [Tur01, MK03]. Both produce high-quality results. The key observation which motivates our choice of per-pixel synthesis is that it only requires *local* information around pixels. This has been previously exploited to synthesize textures *into* a distorted space [YHBZ01, LH06]. In these papers the distortion is modeled by a *Jacobian field*: At every pixel a small matrix describes the local deformation. Our key insight is that per-pixel synthesis will let us synthesize *from* a photograph without having to reconstruct or unfold a piece of surface: We only need to properly define the shape of *local neighborhoods*.

Texture synthesis techniques have been previously used in photographs, for instance to complete holes or to replace textures. Some completion methods either ignore [IP97, CPT03, BVSO03, SYJS05] or do not explicitly model [DCOY03] distortions due to geometry and perspective. These approaches assume that the texture exists in the image at all necessary scales. Because we have no prior knowledge of the distortion in the output, we cannot make such an assumption.

The approach of Ofek et al. [OSRW97] extracts the texture of a known object from a sequence of photographs, taking into account geometry and perspective. However, it is difficult to apply in our context where only a single photograph is available and the geometry is unknown. Synthesizing a texture of similar appearance fortunately only requires partial knowledge about the surfaces present in the image. Guided texture synthesis [HJO*01, Har01] let the user control which regions of the input are used to synthesize regions of the output. Perspective distortion may thus be captured by a gradient in the guidance field. This however strongly restricts the search space. Often the user is asked to identify how square regions get mapped into the final image [IBG03, LLH04, PSK06], typically by overlaying a distorted lattice over the photograph. This is used to re-project a flat synthesis result into the image, giving the illusion it was generated with the appropriate distortion. This has several drawbacks: Manually specifying the lattice can prove difficult in absence of visual clues. Going back from flat to distorted space involves re-sampling, possibly reducing quality and wasting computations by synthesizing at a higher resolution than required in the image. Finally, and most importantly, piecewise flat regions cannot properly capture a curved surface under projection.

Fang et al. [FH06] limit user involvement and estimate the surface normals from the shading. However normals are not sufficient for synthesis from a surface as we also need texture scale and orientation. Since most of the surfaces we deal with are well captured by curved patches, it seemed unnecessary to follow these more complex reconstruction approaches. We instead let the user describe pieces of geometry through a simple, improved user interface.

3. Our approach

Figure 1 depicts the main steps of our approach: Using our interface a user specifies a rough approximation of the scene geometry in the input photograph (a) to model *local* distortions as a Jacobian field (b). Intuitively, the Jacobian field defines two vectors in each pixel telling us where to find neighboring texture elements in the image. The visualization outlines the streamlines of those two principal directions with red and green anisotropic noise patterns obtained from a line integral convolution [CL93]. Next the user selects a region of interest (ROI) through a simple paint interface (c). We use the Jacobian field to analyze the texture in the ROI and synthesize directly *from* distorted space *to* distorted space. For example we can synthesize into the same photograph to remove a window (d).

The key idea for existing anisometric texture synthesis [YHBZ01,LH06] is the comparison of the *distorted* synthesis neighborhood with *regular* neighborhoods in a flat exemplar (Figure 2(c) and 2(b)) searching for the best match. We generalize this idea and directly compare *distorted* synthesis neighborhoods to *distorted* exemplar neighborhoods (Figure 2(c) and 2(a)).

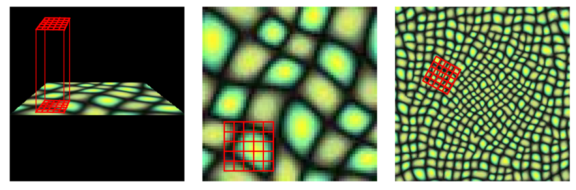

(a) Distorted exemplar space (DES) (b) Flat exemplar (c) Distorted synthesis space (DSS)

Figure 2: *Anisometric synthesis compares distorted synthesis neighborhoods (c) with regular neighborhoods (b). We compare them directly to distorted exemplar neighborhoods (a). Representative neighborhood shapes are outlined in red.*

A key element in understanding our approach is that we only work with two spaces: The distorted exemplar space (DES) – the input – and the distorted synthesis space (DSS) – the output. A flat exemplar *does not exist* in our case. For neighborhood comparison we build regular square neighborhoods expressed in the same frame of reference: The regular neighborhood frame (RNF). While these neighborhoods have a regular, square shape in the RNF they of course have arbitrarily distorted shapes in DES and DSS. We use the Jacobian field to compute the distortion of the neighborhoods and fetch the colors at the appropriate locations around each pixel in the images. This amounts to implicitly flatten local neighborhoods for comparison. Note that it is different from a global flattening of the exemplar which would introduce distortions in the general case.

3.1. User interface

Our synthesis scheme assumes we have a Jacobian field to model the distortions of the input texture. Obtaining such a field for a given photograph is challenging, since it results from the combination of several factors: Projection onto the image plane, surface geometry, texture orientation and scale.

In practice, many interesting textured surfaces are planar or curved – with a curvature that is small compared to the scale of the texture. This has been exploited by several approaches relying on a user interface [LLH04,DTM96, IBG03,PSK06]: The user specifies the distortion of the texture by placing a few primitives in the view. However we have seen that these interfaces have disadvantages when dealing with curved surfaces. Hence we propose a new improved interface for this task. A key issue is that the primitives should have a reasonably low number or degrees of freedom to allow easy interaction, yet be powerful enough to describe planar and curved surfaces under projection.

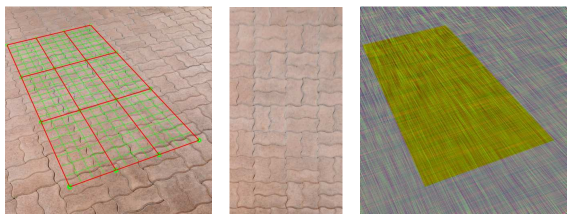

(a) Photograph (DES) (b) Flat (c) Jacobian field

Figure 3: *In the planar case we use the 2D homography between (a) and (b) to define a Jacobian field J_{DES} (c).*

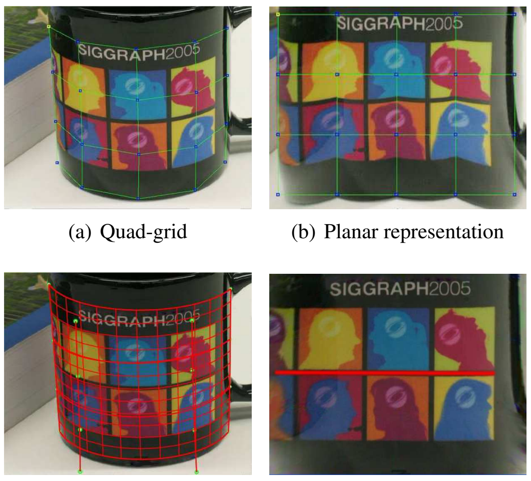
Planar geometry: In the planar case we build on the 2D homography approach of Pavić et al. [PSK06]. The user places four corner points to define a 2D homography between the DES and a flat rectangle. We use a Direct Linear Transformation (DLT) to obtain a homogeneous 3×3 projection matrix M_{DLT} between the two polygons. However, while Pavić et al. use it to flatten the texture and operate in flat space, we do the opposite. We use it to transform the identity frame from the flat space into the DES. As shown in Figure 3(c) these projected vectors define the Jacobian field used as input for our synthesis from distorted space algorithm.

Curved geometry: Unfortunately, the piecewise flat approximation of curved surfaces causes high order discontinuities like the 'zig-zag' distortions in Figure 4(b), which are bad for synthesis as they define wrong neighborhoods. Previous UIs attempt to lessen such distortions using very fine grids [LLH04, PSK06]. However, placing that many nodes is tedious for the user, especially for textures that provide few visible clues for their placement.

Fortunately, defining smooth curved surfaces with a few control points (CPs) is exactly what Bézier curves were designed for. We are most interested in *rational* Bézier curves, which have an additional user controllable weight w_i at each CP $\mathbf{b_i}$:

$$x(t) = \frac{\sum_{t=0}^{n} w_i \mathbf{b_i} B_i^n(t)}{\sum_{t=0}^{n} w_i B_i^n(t)} \qquad (1)$$

While interaction with a 2D rational Bézier curve is a 2D process – CPs are moved around and weights are modified until the resulting curve has the desired shape – it is well known, that the 2D CPs $\mathbf{b_i}$ and their weights w_i in fact define 3D CPs $[\; w_i\mathbf{b_i} \quad w_i \;]^T$ and a 3D Bézier curve that is projected onto the plane $w = 1$ [Far02]. This is exactly our scenario, making rational Bézier patches the ideal tool to describe curved objects in photographs as shown in Figure 4.



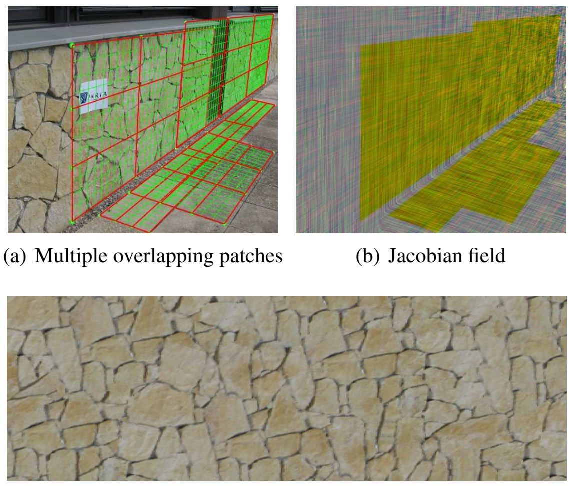
(a) Quad-grid (b) Planar representation

(c) Rational Bézier patch (d) Flat parameter space

Figure 4: *Comparison with [PSK06]: Piecewise flat approximation (a) causes C^1 discontinuities on curved surfaces (b). Rational Bézier patches (c) capture them correctly (d). Images (a) and (b) with permission of [PSK06].*

In general a Bézier patch will describe a non-developable surface. To obtain a Jacobian field we compute the derivatives of each patch *in homogeneous coordinates* and normalize them before projection onto the image plane.

Because our approach works directly in distorted space, it avoids resampling of the exemplar into a flat image and is capable of handling non-flattenable curved surfaces like the apple in Figure 17. Additionally, it enables the use of multiple patches in a very robust way: As we do not flatten the texture, we only have discontinuities in the Jacobian field, i.e. the first derivative. C^0 continuity is always maintained and no relaxation algorithm [PSK06] needs to be applied to recover from imprecise user input. Our patches can overlap, be disjoint, and may in general not perfectly match. Simply averaging the Jacobians of overlapping patches and closing gaps by extrapolating with a pull-push algorithm [GGSC96] suffices for texture synthesis as shown in Figure 5.


(a) Multiple overlapping patches (b) Jacobian field

(c) Synthesis result

Figure 5: *The user inputs multiple overlapping patches (a). Still the Jacobian field shows no discontinuities across patch borders (b) and a convincing texture is synthesized (c). Note that a single patch (four interactions) would suffice here.*

In practice, we ask the user to arrange CPs and adjust weights until an overlaid grid roughly matches the distortion in the image. A good initialization is essential in reducing interaction time. Hence, we start with a rough planar approximation of the surface. We take four user specified points, perform a DLT, obtain a projection matrix M_{DLT} and use it for the initial placement of the CPs at regular positions in the flat parameter space. Our key insight is that M_{DLT} delivers the exemplar position of the CPs in *homogenous coordinates*. While most applications just perform de-homogenisation, we realize that the homogenous components are exactly the initial weights $w_{i,j}$ for our rational Bézier patch. In addition, assuming square pixels in the camera and a square parameter space, we can extract the aspect ratio of the user specified *rectangle* and adjust the subdivision of our overlaid grid like in Figure 3(a). In the final interface, the user moves the four corners of the patch until the overlaid grid gives the same feeling of perspective as the photograph. If the user wants to describe a plane, interaction is complete. If a curved surface is desired, the user has to move the individual CPs and in rare cases adjust the CP weights.

In our experiments we found that rational Bézier patches of degree three, i.e. *cubic rational Bézier patches*, were most suitable for our task. They are a good trade-off between complexity for the user – $4 \times 4 = 16$ CPs are already a lot – and the possible power of expression.

It is important to note that the matrix M_{DLT} is only unique up to a scaling factor λ. While this is irrelevant for dehomogenisation, it may lead to unpredictable UI behavior with our approach and is especially confusing if M_{DLT} produces negative weights, i.e. some CPs are behind the camera. We solve this pragmatically by scaling M_{DLT} so that all CPs have positive weight and the closest edge has a user specified length. The latter also lets us define multiple patches while ensuring a consistent scaling.

Region of interest: The last step for the user is to define which areas should be used for synthesis. This is done on a per-pixel basis with a simple paint interface like Figure 1(c).

3.2. Exemplar analysis

While our ideas do not depend on a particular synthesis scheme, we chose to base our work on the recent algorithm of Lefebvre and Hoppe [LH06]. Like many synthesis algorithms, it splits the workload into a slow analysis step and a fast synthesis step [ZG02]. Most of the changes we made are related to exemplar analysis. In order of processing, we adapt the following steps:

- We compute a *distorted feature distance* for structured (i.e. non stochastic) textures.
- We compute a *sparse distorted Gaussian stack*: A multiscale description of the input texture, discarding areas not usable due to lack of detail.
- We fetch the *distorted neighborhoods*.

We describe each of these steps in the following.

Distorted feature distance: An important information to add to the neighborhoods of structured exemplars is the feature distance (FD) [LH06]. At each pixel it describes the distance to the border of the current feature, hence guiding the synthesis algorithm. It is obtained from a binary feature mask by computing the signed euclidean distance transformation. The example mask in Figure 6 was obtained by manually editing the result of an edge-detection filter.

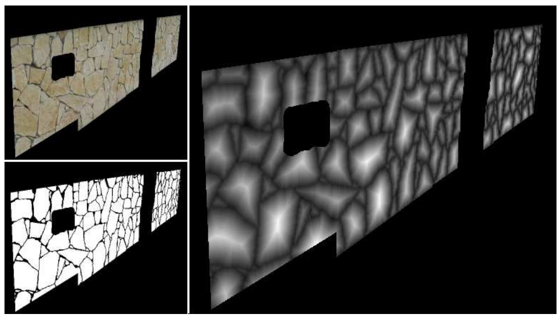
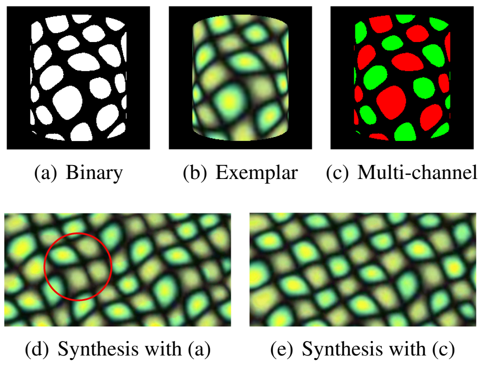


Figure 6: *For a given photograph (top) the user defines a binary feature mask (bottom). Using the Jacobian field, we compute a distorted feature distance (right).*

We adapt an efficient algorithm [Dan80] to operate through the distortion field. This only involves a simple modification: The Jacobian is taken into account when updating per-pixel distance labels from their neighbors. The result is the distorted feature distance illustrated in Figure 6. Note that while the distant stones cover less pixels than the stones in the foreground, they have similar size in reality and indeed comparable feature distances.

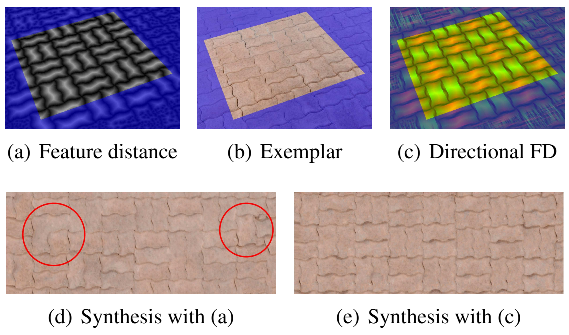


(a) Binary (b) Exemplar (c) Multi-channel

(d) Synthesis with (a) (e) Synthesis with (c)

Figure 7: *Simple feature distance fails on this alternating pattern (d). Multiple channels succeed (e).*

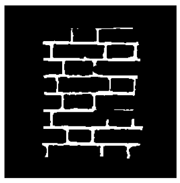
We found that feature distance fails to improve synthesis in certain cases: Interleaved features of different kind and anisotropic textures. We improve the first by using multiple channels to capture the distance to different features. This is inspired by the texture-by-number approach described by Hertzman et al. [HJO*01]. Note how we are able to preserve the alternating pattern of bright and dull dots in Figure 7(e). We address the second issue by encoding the distance *along certain directions* in multiple channels. Indeed, the key observation is that standard feature distance fails to capture asymmetric features like the bricks in Figure 9 because it encodes distance to the *closest* feature only. Instead we encode the distance to the closest feature *along multiple directions*, typically the two principal directions defined by the Jacobian field. As shown in Figure 8 and 9, quality is significantly improved by this simple extension.



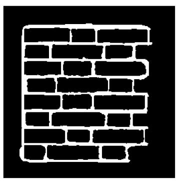
(a) Feature distance (b) Exemplar (c) Directional FD

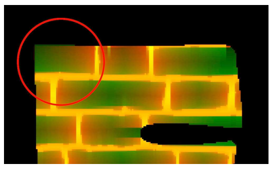
(d) Synthesis with (a) (e) Synthesis with (c)

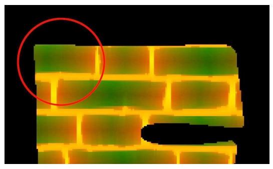
Figure 8: *Standard Feature Distance (a) produces artifacts with anisotropic textures (d). Directional Feature Distance (c) solves this problem (e).*

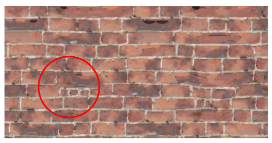

(a) Standard mask

(b) Exemplar


(c) Extended mask


(d) Standard directional FD


(e) Extended directional FD


(f) Synthesis with plain FD

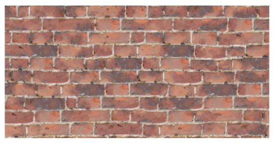
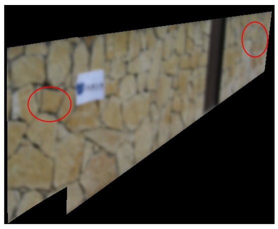

(g) Synthesis with extended FD

Figure 9: *Defining a feature mask only for the ROI introduces errors into the FD (d) and results in artifacts (f). Extending the mask (c) avoids these errors and artifacts (g).*

A last point we want to emphasize is the need to define the feature mask beyond the boundaries of the ROI. Without this extension, the feature distance within the circled brick in Figure 9(d) becomes arbitrarily large and introduces wrong information into the analysis. This can be corrected by manually extending the feature mask beyond the ROI boundary, as shown in Figure 9(c).

Sparse distorted Gaussian stack: Like most texture synthesis algorithms, ours relies on a multi-scale process. The input image is filtered at different scales and the texture is synthesized progressively, using coarse results as a guide to synthesize finer results.

In order to follow this approach, we need to filter the texture in the input image. Our goal is to define a Gaussian stack [LH05]: An array of images containing successively low-pass filtered versions of the texture. We face two main challenges: First, we have to filter in distorted space. As Figure 10(a) illustrates, convolving the complete image with a fixed size Gaussian kernel is incorrect in the context of distorted exemplars. The input texture is sampled non-uniformly and an image space convolution does not correspond to a convolution in the flat texture space. Second, the image is likely to contain a texture already filtered at different scales due to geometry and perspective. We have to detect and discard pixels that are already low-pass filtered by the capture process and do not provide enough information for the finer levels of the stack.


(a) Low-pass filtered

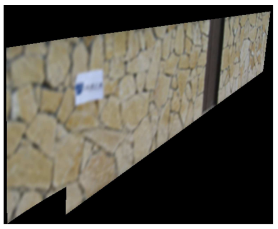

(b) Local low-pass filtered

Figure 10: *Low-pass filtering the complete photograph is incorrect: distant areas are overly blurred (a). Our local filtering technique solves this issue (b).*

Our *local filtering* technique relies on the separability of the Gaussian kernel and performs two successive line integral convolutions [CL93] along the local streamlines defined by the two principal directions of the Jacobian field. As shown in Figure 10(b) the result is a low-pass filter that is non-uniform and anisotropic in image space – areas close to the viewpoint are blurred stronger than distant areas. Yet it is uniform with respect to the texture features: The mortar remains visible over the complete wall. It is important to note that this method works for non-developable surfaces as well. This approach may lead to artifacts at the border of the filtered region. However, this is not a problem for texture synthesis since neighborhoods partially outside the ROI will be ignored.

To detect whether the texture is already filtered in the input image, we again rely on the Jacobian field. Our way is similar to [IBG03], however we use a different texture quality metric. The key observation is that the determinant $\det J_{xy}$ equals the area in image space spanned by the two vectors of the Jacobian J_{xy}. We re-scale the Jacobian field so that the largest occurring determinant of J' equals a one pixel area. This corresponds to the highest frequency detail captured by the image and defines the size of one 'texture element'. The determinant of the inverse, $\det J'^{-1}_{xy}$, is the number of texture elements per pixel and pixels containing multiple texture elements have been low-pass filtered in the photograph.

Following this idea, we want to detect whether a pixel should be included in a particular stack level SL. By definition, at level 0 one texture element maps to one pixel. At level 1 four texture elements (2×2) map to one pixel. More generally, we obtain the stack level from the Jacobian as:

$$SL_{xy} = \log_4 \left(\det J'^{-1}_{xy} \right) \qquad (2)$$

When building the final sparse distorted Gaussian Stack we only keep pixels that have an SL lower or equal to the current level of the stack, as all other pixels lack the necessary high frequency content. The resulting stack is shown Figure 11. Notice how finer levels lack information in areas that are too far away from the viewpoint.

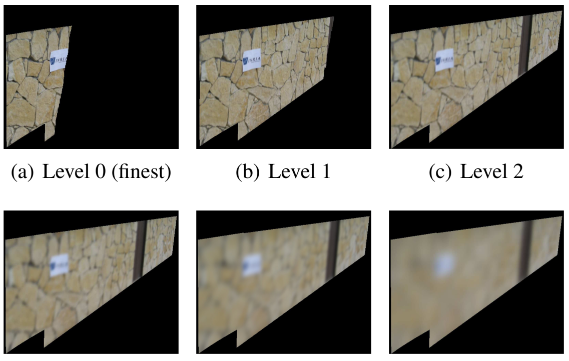

(a) Level 0 (finest) (b) Level 1 (c) Level 2
(d) Level 3 (e) Level 4 (f) Level 5 (coarsest)

Figure 11: *The combination of local filtering and stack level technique results in a sparse distorted Gaussian stack. Only the first 6 levels are shown.*

Distorted neighborhoods: The key element of pixel-based texture synthesis is the search for the best matching exemplar neighborhood. Similarly to algorithms synthesizing into distorted space, we use a relative local neighborhood definition shown in Figure 12(a). In our terminology this regular 5×5 reference neighborhood is defined in RNF using offset vectors from the centre to the neighboring texture elements.

We build the neighborhoods of the pixels in DES and DSS one entry at a time. For each entry in the neighborhood, we start from the center and march along the offset, following the Jacobian field as illustrated in Figure 12(b). We always advance at most by one pixel in distorted space, until the entire offset has been processed. If one neighbor falls outside the ROI, the neighborhood is only partially valid – which hinders pre-computation – and hence it is ignored.

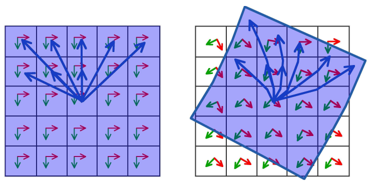

(a) Relative definition (b) Distorted nbh.

Figure 12: *The offsets of the relative neighborhood definition in RNF (a) are distorted by the Jacobian field of the DES and the DSS to obtain distorted neighborhoods (b).*

3.3. Texture synthesis

At this point we have seen all the required elements to adapt a standard per-pixel synthesis algorithm to our approach. The main changes to the synthesis part, which we describe below, concern the construction of the neighborhoods, handling of the exemplar sparsity, and antialiased synthesis when targeting a distorted space.

Neighborhoods: Neighborhoods are built from the DSS during synthesis in the same way they are constructed from the DES during the analysis. Neighborhoods are all expressed in RNF and are directly compared to the DES neighborhoods built during the analysis step.

Border jumps: One feature of the synthesis scheme we use is that it synthesizes exemplar coordinates. Once a result is obtained at a given resolution, the resolution is increased by coordinate *upsampling*: Every synthesized coordinate is replaced by the coordinates of its four children in the next finer level of the analysis stack. Neighborhood matching is then performed on this upsampled result.

For a flat square exemplar, upsampling is a trivial computation on the coordinates. However, when using a sparse exemplar, the computed position may fall outside the user-defined ROI. We solve this issue by marching along the offset between parent and child coordinate, similarly to the march performed for gathering a neighborhood. Care has to be taken to correctly transform the offset from DSS to DES. Whenever we encounter a neighborhood that does not completely lie inside the ROI, the walk will be 'teleported' to a valid location. The target of this *border jump* is precomputed to be the neighborhood entirely inside the ROI most similar to the partially defined neighborhood at the boundary. A nice implementation detail is that we can hide this mechanism inside the k-coherence search [TZL*02] used by most texture synthesis algorithms. Figure 15 shows a synthesis result obtained from a sparse exemplar.

Antialiased synthesis: For synthesis into a space with distortions that model a scaling, the scale of texture features and neighbourhood size are decoupled from the level of the synthesis pyramid. Strong aliasing effects may occur. Figure 13 shows an example where the Jacobian field for DSS is a scaling ramp from 1 to 0. Aliasing appears as $\det J$ approaches 0, implying that an increasingly large number of texture elements map to a single pixel. Instead of showing the average color of these texture elements, the pixel displays only one of them, producing aliasing. Similarly to the mechanism described Section 3.2, we compute the number of texture elements in each pixel of the synthesis pyramid. Note that at coarser levels, pixels enclose the texture elements of their children. We then compute the stack level to be used at each pixel using this number.

4. Results and applications

For all the examples shown in the paper the user interaction to describe the distortions took between 10 seconds and two minutes (Figure 1). Feature masks were obtained using an edge-detector – a Difference of Gaussian filter – and thresholding. Except for the artificial examples in Figures 7 and 9 the feature masks naturally extend outside of the ROI. In Figures 6 and 8 noise artifacts were manually removed. The clusters in Figure 7(c) were created with the GIMP filler tool.

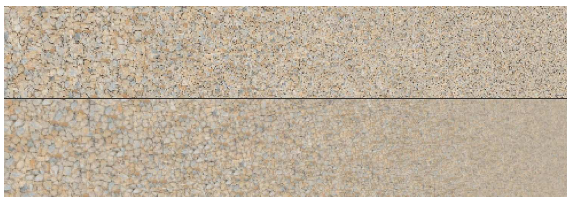


Figure 13: *Synthesis using a* $[1..0]$ *ramp as Jacobian field. Top: Ordinary synthesis; bottom: Antialiased synthesis. Texture from Figure 17(a).*

Analysis time for the results displayed in this paper ranges from 2 to 10 minutes on a single core Intel P4 3.2 GHz. Synthesis time ranges from 1 to 5 minutes depending on the synthesized area size. Both analysis and synthesis cost grow linearly in number of pixels. While this is an acceptable speed, it is slow compared to state-of-the-art synthesis performance. Most of the cost currently comes from neighborhood gathering. However, these measurements are for our CPU implementation. None of our changes prevent the GPU implementation of [LH06], and we believe much faster synthesis can be achieved if necessary.

The naive approach to synthesize from a photograph would be to ignore texture distortions, cut out a square piece of the image and use it for synthesis. As illustrated in Figure 14, this fails if the image contains a distorted surface. For the same example we ask the user to specify a rough approximation of the underlying geometry. After less then 30 seconds of interaction, our approach synthesises a much more faithful result.

As illustrated in Figure 15, we can synthesize from sparse exemplars and are able to perform synthesis under severe distortion while maintaining synthesis quality. Notice that synthesized content is properly antialiased where features become small. Figure 16 shows a challenging case of low-pass filtering during image capture. The 128^2 example was generated by rendering a polygon with a checkerboard texture, using hardware anisotropic filtering. Even though only few unfiltered checkerboard cells exist in the foreground, our scheme synthesizes an almost perfect checkerboard.

A first straight forward application of our approach is to prepare flat, square, toroidal textures from a photograph. These flat samples can later be used as input exemplars for any 'flat' synthesis scheme.

Another application is the completion of textures in photographs. We define a ROI for the texture, a target area and use the overlapping area as constraint. This is achieved by fixing the value of constraining pixels in the synthesis pyramid. As illustrated in Figure 17, we successfully complete textures on both planar and non-flattenable surfaces. This procedure can also be used to handle partial occlusions for subsequent processing.

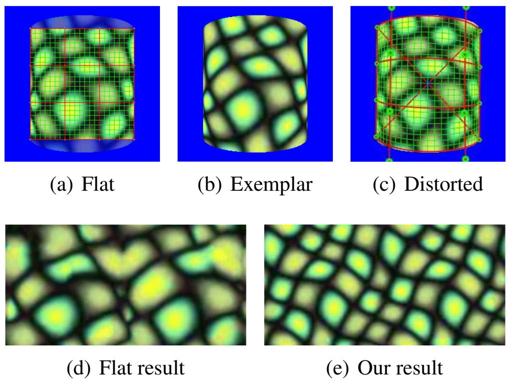


Figure 14: *Photograph of a curved exemplar (b): Using a flat portion of the image (a) synthesis fails (d). Modeling the distortions with a rational Bézier patch (c) synthesis works well (e). Note that (d) and (e) have the same resolution.*

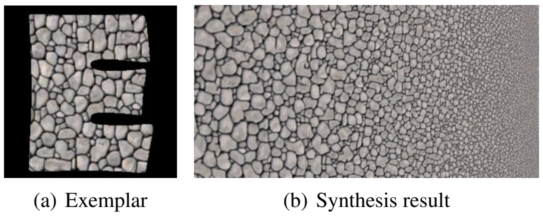


Figure 15: *Anisometric, antialiased synthesis of pebbles from a sparse exemplar.*

As we synthesize pixel coordinates, we can easily verify that the algorithm is not limited to using pixels from areas with similar distortion in the input. In Figure 18, pixels from both the foreground and the background are used for synthesis in the back. A texture-by-number approach [HJO*01] would miss these opportunities by over-constraining the search space. Note that synthesis in the front does not use pixels from the back as they do not contain enough details.

Another strength of our approach is that we are not limited to the content of a single image. We can synthesize arbitrary amounts of texture from one image into another.

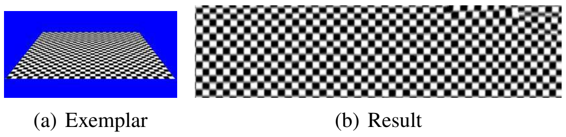


Figure 16: *Few unfiltered checkerboard cells exist in the foreground (a), still our scheme synthesizes an almost perfect checkerboard (b).*

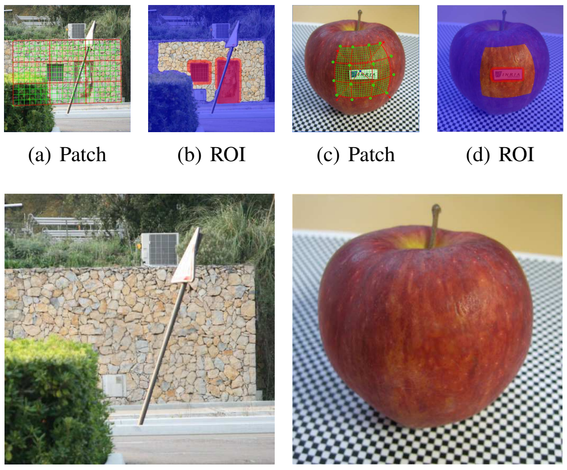


Figure 17: *The user specifies a patch for distortion (a, d), a ROI and a target area (b, e) to complete an image (c, f).*

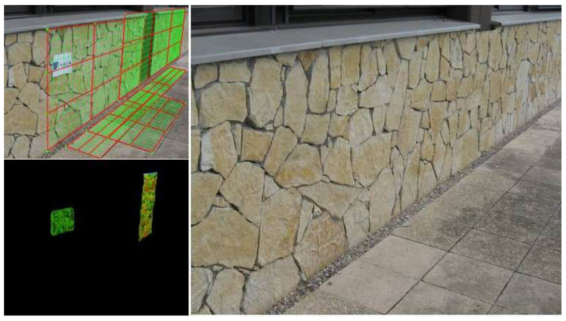


Figure 18: *The color map in the bottom left encodes in red and green the position the texture was taken from in synthesized areas. Green is foreground, red is background. We use all available textures as long as it has enough details.*

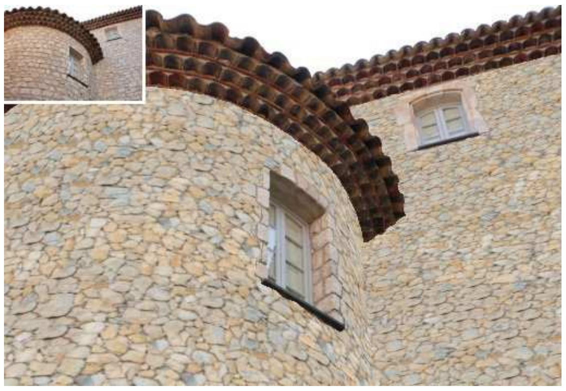


Figure 19: *We can use the texture from Figure 17(b) and place it onto the geometry of Figure 1 (shown inset).*

Figure 19 shows an example where the synthesized texture did not exist anywhere with the appropriate distortion in the input. Still, since we explicitly model distortion, our result follows the surface nicely. Note however that we focus on texture synthesis and did not try to preserve shading in the target image. This is, of course, an interesting avenue for future research.

Our scheme produces lower quality results in some extreme situations. For instance, using sparse exemplars containing large features like the ‘brick wall E’ in Figure 9 will lead to artifacts as too few complete neighborhoods are available at coarse resolution. Similarly if too few pixels contain high resolution information in the input, it is difficult to synthesize large high-resolution areas in the output, due to lack of data.

5. Conclusion

We introduced a method to synthesize new textures using any surface from any photograph as input.

We let the user roughly describe the surfaces present in the image by combining rational Bézier patches through a simple interface. Rational Bézier patches are particularly well suited for this task as they directly correspond to 3D Bézier patches projected onto a plane. Thanks to a good initialization of the control points, placing new patches in the image is easy for the user. Our interface outputs a Jacobian field which locally defines, in each pixel, the distortion of the texture in the image.

Using only this local information as input, we build upon an existing state-of-the-art synthesis algorithm to synthesize from a sparse, distorted input texture. We showed how to adapt each key synthesis step: Computation of a distorted feature distance to guide synthesis of structured patterns, multi-scale analysis of the texture content, gathering of distorted neighborhoods, border jumps for upsampling in sparse exemplars and antialiased synthesis.

The end result is a per-pixel synthesis algorithm which can extract a texture from any surface in a photograph, and synthesize a new texture under arbitrary distortion. Applications range from texture synthesis to texture completion.

Possible future directions of research include:

- Using neighborhood approximations similarly to [LH06] and targeting a GPU implementation for real-time synthesis from distorted input.
- Respect shading for better texture replacement.
- Using an interface similar to the one shown by Fang et al. [FH07] to obtain curvelinear coordinates. One challenge will be to capture perspective effects.
- Replacing the user interface with a shape from shading approach or using a few markers in the image. One challenge is to determine the texture scale and orientation from the photograph.

References

[BVSO03] BERTALMIO M., VESE M., SAPIRO G., OSHER S.: Simultaneous structure and texture image inpainting. In *IEEE Transactions on Image Processing* (2003), vol. 12, pp. 882–889.

[CL93] CABRAL B., LEEDOM L. C.: Imaging vector fields using line integral convolution. In *Proceedings of ACM SIGGRAPH* (1993), pp. 263–270.

[CPT03] CRIMINISI A., PEREZ P., TOYAMA K.: Object removal by exemplar-based inpainting. *Computer Vision and Pattern Recognition 2* (2003), pp. 721–728.

[Dan80] DANIELSSON P. E.: Euclidean distance mapping. *Computer Graphics and Image Processing 14* (1980), pp. 227–248.

[DCOY03] DRORI I., COHEN-OR D., YESHURUN H.: Fragment-based image completion. *ACM Transactions on Graphics 22*, 3 (2003), pp. 303–312.

[DTM96] DEBEVEC P. E., TAYLOR C. J., MALIK J.: Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM, pp. 11–20.

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. In *Proceedings of ACM SIGGRAPH* (2001), pp. 341–346.

[EL99] EFROS A. A., LEUNG T. K.: Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision* (Corfu, Greece, September 1999), pp. 1033–1038.

[Far02] FARIN G.: *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[FH06] FANG H., HART J. C.: Rototexture: Automated tools for texturing raw video. *IEEE Transactions on Visualization and Computer Graphics 12*, 6 (2006), pp. 1580–1589.

[FH07] FANG H., HART J. C.: Detail preserving shape deformation in image editing. *ACM Transactions on Graphics 26*, 3 (2007), 12.

[GGSC96] GORTLER S. J., GRZESZCZUK R., SZELISKI R., COHEN M. F.: The lumigraph. *Computer Graphics 30*, Annual Conference Series (1996), 43–54.

[Har01] HARRISON P.: A non-hierarchical procedure for re-synthesis of complex textures. In *WSCG Conference Proceedings* (2001).

[HJO*01] HERTZMANN A., JACOBS C. E., OLIVER N., CURLESS B., SALESIN D. H.: Image analogies. In *Proceedings of ACM SIGGRAPH* (2001), pp. 327–340.

[IBG03] ISMERT R. M., BALA K., GREENBERG D. P.: Detail synthesis for image-based texturing. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics* (2003), pp. 171–175.

[IP97] IGEHY H., PEREIRA L.: Image replacement through texture synthesis. In *Proceedings of the International Conference on Image Processing* (1997), p. 186.

[KSE*03] KWATRA V., SCHÖDL A., ESSA I., TURK G., BOBICK A.: Graphcut textures: Image and video synthesis using graph cuts. *Proceedings of ACM SIGGRAPH* (2003), pp. 277–286.

[LH05] LEFEBVRE S., HOPPE H.: Parallel controllable texture synthesis. *ACM Transactions on Graphics 24*, 3 (2005), pp. 777–786.

[LH06] LEFEBVRE S., HOPPE H.: Appearance-space texture synthesis. *ACM Transactions on Graphics 25*, 3 (2006), pp. 541–548.

[LLH04] LIU Y., LIN W.-C., HAYS J.: Near-regular texture analysis and manipulation. *ACM Transactions on Graphics 23*, 3 (2004), pp. 368–376.

[MK03] MAGDA S., KRIEGMAN D.: Fast texture synthesis on arbitrary meshes. In *Proceedings of the Eurographics workshop on Rendering* (2003), pp. 82–89.

[OSRW97] OFEK E., SHILAT E., RAPPOPORT A., WERMAN M.: Multiresolution textures from image sequences. *IEEE Computer Graphics and Applications 17*, 2 (1997), 18–29.

[PSK06] PAVIĆ D., SCHÖNEFELD V., KOBBELT L.: Interactive image completion with perspective correction. *The Visual Computer 22*, 9 (2006), pp. 671–681.

[SYJS05] SUN J., YUAN L., JIA J., SHUM H.-Y.: Image completion with structure propagation. In *Proceedings of ACM SIGGRAPH* (2005), pp. 861–868.

[Tur01] TURK G.: Texture synthesis on surfaces. *Proceedings of ACM SIGGRAPH* (2001), pp. 347–354.

[TZL*02] TONG X., ZHANG J., LIU L., WANG X., GUO B., SHUM H.-Y.: Synthesis of bidirectional texture functions on arbitrary surfaces. In *Proceedings of ACM SIGGRAPH* (2002), pp. 665–672.

[WL00] WEI L.-Y., LEVOY M.: Fast texture synthesis using tree-structured vector quantization. In *Proceedings of ACM SIGGRAPH* (2000), pp. 479–488.

[YHBZ01] YING L., HERTZMANN A., BIERMANN H., ZORIN D.: Texture and shape synthesis on surfaces. In *Proceedings of the Eurographics Workshop on Rendering Techniques* (2001), pp. 301–312.

[ZG02] ZELINKA S., GARLAND M.: Towards real-time texture synthesis with the jump map. In *Proceedings of the Eurographics Workshop on Rendering* (2002), pp. 99–104.